

Testes Unitários no desenvolvimento iOS

A importância de testar seu aplicativo iOS

Introdução

O pior cenário para o desenvolvedor iOS é o cliente baixar seu aplicativo e, durante o seu uso, ou mesmo logo de início, o seu aplicativo *quebrar* (fechar abruptamente). Alguns usuários, mais tolerantes, irão tentar novamente, mas outros não. O seu app, que você passou meses desenvolvendo, será sumariamente excluído do dispositivo daquele cliente, e ele provavelmente nunca mais irá tentar baixá-lo. Centenas de horas de trabalho desperdiçadas em poucos segundos.

Como essa situação poderia ter sido evitada, ou pelo menos minimizada?

Testes como garantia de qualidade

Ok, você estudou, fez cursos, sofreu, passou madrugadas acordado, superou vários obstáculos e agora está preparado para colocar seu 1º app na loja. Mas, será que seu código está bem escrito? Será que todos os cenários previstos estão considerados? Será que o app está preparado para lidar com uma situação inesperada, como um texto vazio, um campo nulo ou ausência de valores onde eles deveriam existir?

Estamos falando de qualidade na construção do código fonte, e o mecanismo para garantir essa qualidade são os testes. Eles não são obrigatórios, mas a presença deles poderia evitar situações catastróficas para o app, que poderiam ser evitadas com um teste simples se uma função está retornando o resultado esperado.

Os testes, mais conhecidos como testes unitários, existem desde os primórdios da programação, mas muita gente não os utiliza. A ideia é que você escreva testes para testar suas unidades de código, como uma função, uma classe ou um protocolo.

O problema é que dá trabalho escrever testes unitários, pois é como se a lógica tivesse que ser reescrita, mas sob uma outra ótica. E o mais importante; os testes unitários são de responsabilidade do próprio desenvolvedor. Não há como fugir dessa responsabilidade. É para você, desenvolvedor, que vão apontar o dedo no 1º problema que ocorrer em produção, e você vai ouvir aquela frase célebre: "Mas você não *testou* essa funcionalidade?".

E se você é um desenvolvedor independente, que trabalha no quarto de sua casa, e acha que está livre de toda a pressão corporativa, não se iluda. Quem irá te julgar, ou ao seu app, é o cliente final, e de certa forma isso é até pior do que julgado pelo seu chefe, pois são eles, os clientes, que irão utilizar o seu aplicativo, ou não.

Os tipos de Testes

Espero tê-los assustado o suficiente para continuar a leitura. Mas a boa notícia é que escrever testes não é tão complicado. O maior desafio é incorporar essa boa prática ao seu dia-a-dia de desenvolvedor.

Vamos aos tipos de testes.

Testes Unitários

São os testes básicos de unidade, ou seja, validar a funcionalidade da menor unidade de código de seu aplicativo. Normalmente esses elementos são o que conhecemos como funções - as `func` em Swift ou métodos, em Objective-C.

Por exemplo, a função de soma de seu app de calculadora deve retornar o valor exato da soma dos parâmetros passados a ela, e é justamente isso que o teste irá verificar.

Os testes unitários são de responsabilidade do desenvolvedor e normalmente, nas empresas, são considerados como indicadores de qualidade. O percentual de cobertura de testes unitários é um dos principais indicadores de qualidade de desenvolvimento do time envolvido no projeto, e normalmente não se aceita nada menos que 70% de cobertura.

Testes de Interface

Apesar de não tão comuns como os testes unitários, os testes de interface vem ganhando importância dentro dos times de desenvolvimento, pois eles verificam situações que os testes unitários não conseguem validar. Podemos considerar que se tratam mais de testes de comportamento do aplicativo, e não de unidade de código.

Por exemplo, se um botão que leva à exibição de uma tela de detalhes estiver desabilitado, a navegação não deve ocorrer, mesmo que o usuário clique no botão para exibir os detalhes. Porém, se o botão estiver habilitado, a navegação para a tela de detalhes deverá, sim, ocorrer. Portanto serão construídos dois testes de interface para validar esses comportamentos.

Testes Automatizados

Os testes automatizados atuam sobre a aplicação compilada como um todo. É como se os testes unitários e de interface estivessem no escopo dos testes de caixa branca, segundo o termo acadêmico, e os testes automatizados estivessem no escopo de testes de caixa preta.

Normalmente eles exigem ferramentas de terceiros específicas para essa finalidade, e profissionais de qualidade que estejam aptos a construir os testes, segundo as funcionalidades do aplicativo como um todo.

Numa cadeia de desenvolvimento onde trabalha sobre os preceitos da metodologia ágil, equipes estruturadas de **DevOps** (desenvolvimento e operações) normalmente implementam suas próprias soluções, que sejam adequados ao ambiente de desenvolvimento em que atuam.

TDD - Test Driven Development

O TDD não é bem um tipo de teste e sim uma metodologia de trabalho, que prega que os testes devem ser escritos antes mesmo da lógica em si, pois desta forma, as funções já nasceriam *testadas* e *validadas*.

É um conceito polêmico e bastante discutido na comunidade de desenvolvedores em geral.

Não abordaremos TDD em nosso curso, mas caso queira uma introdução sobre o assunto, segue o link de uma boa introdução ao tema:

<http://initwithstyle.net/2015/11/tdd-in-swift-playgrounds/>

O que o nosso curso irá contemplar

Em nosso curso, iremos contemplar os **testes unitários**.

Precisaremos apenas do Xcode e nada mais, pois utilizaremos o framework nativo **XCTest**, que vem evoluindo a cada versão.

Pré-requisitos

Como pré-requisitos do curso você deve saber os fundamentos da linguagem Swift, e ter conhecimentos básicos de design de aplicativos e conceitos de programação em geral.

O projeto

Em nosso curso, iremos trabalhar com o projeto a seguir, que será construído passo a passo, com o apoio do instrutor:

- **testes-unitarios** - que visa conhecer, entender e aplicar o conceito de Testes Unitários.

Testes Unitários

A partir deste ponto, a apostila informará os procedimentos passo a passo para a criação, desenvolvimento e testes (sem trocadilho) do projeto de testes unitários.

Criando o Projeto

Crie um novo projeto: **testes-unitarios**

- Escolha o Template *Single View Application*
- Na tela de opções, selecione **Include Unit Tests**

Choose options for your new project:

Product Name: testes-unitarios

Team: Carlos Savi

Organization Name: Carlos A Savi

Organization Identifier: com.saviT

Bundle Identifier: com.saviT.testes-unitarios

Language: Swift

Devices: Universal

☐ Use Core Data

☒ Include Unit Tests

☐ Include UI Tests

Cancel Previous Next

Configuração do Projeto

Observe que, ao selecionar a opção de inclusão de testes unitários, o Xcode criou uma nova pasta **testes-unitariosTests**, com a classe **testes_unitariosTests.swift**.

Ao abrir a classe, podemos notar o seguinte:

- Foi importado o framework XCTest, que fornece o motor de testes para o aplicativo
- A instrução `@testable import testes_unitarios`, que indica o projeto foco dos testes unitários.
- Vemos também que o projeto já vem com alguns templates
- Iremos manter as os métodos `setUp()` e `tearDown()`, que atuam sobre todos os testes, mas podemos remover `testExample()` e `testPerformanceExample()`, pois criaremos os nossos próprios testes.
- A classe ficará dessa forma então:

```
import XCTest
@testable import testes_unitarios

class testes_unitariosTests: XCTestCase {

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of each
        test method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each
        test method in the class.
    }
}
```

```

        super.tearDown()
    }

}

```

Criando os métodos a seres testados

A partir de agora, iremos utilizar uma boa prática que prega que o desenvolvedor, a cada método criado, deve criar o teste unitário correspondente. Cria uma FUNÇÃO, cria o TESTE, e assim sucessivamente. Essa prática evita que os testes sejam deixados para o final do desenvolvimento, ou ainda que sejam esquecidos e deixados para um segundo momento.

Criando a primeira função

- Selecione a classe **ViewController.swift**
- Crie a função a seguir, cuja finalidade é verificar se um número é par, e retornar *true*, caso seja:

```

// MARK: - Funções que serão testadas

// 1 - Função simples
func isNumberEven(num: Int) -> Bool {
    if num % 2 == 0 {
        return true
    } else {
        return false
    }
}

```

Criando o teste

A criação de cada teste deve ser planejada pelo desenvolvedor. Para cada caso, o que precisa ser testado? Como testar essa primeira função?

Como é uma função que retorna um valor booleano, devemos portanto testar os dois retornos possíveis - *true* e *false*. Verdadeiro quando a função receber um número par e falso quando receber um número ímpar.

Para isso, utilizaremos os métodos de teste nativos `XCTAssertFalse` e `XCTAssertTrue`.

Se você posicionar o cursor sobre o nome do framework - **XCTAssert**, e digitar a sequência de teclas command+clique, você verá todas as possibilidades de testes.

Vamos retornar à classe **testes_unitariosTests.swift** e escrever nosso primeiro teste. O *style guide* de testes pede que as funções de teste sempre iniciem pelo prefixo **test**:

- Ao final da classe, crie a seguinte função:

```
// Testes Unitários
```

```
func testIsNumberEven() {
```

```
    // Necessário para instanciar a View Controller
```

```
    let viewController = ViewController()
```

```
    // Fail Test
```

```
    let odd = 9
```

```
    XCTAssertFalse(viewController.isNumberEven(num: odd))
```

```
    // True Test
```

```
    let even = 12
```

```
    XCTAssertTrue(viewController.isNumberEven(num: even))
```

```
}
```

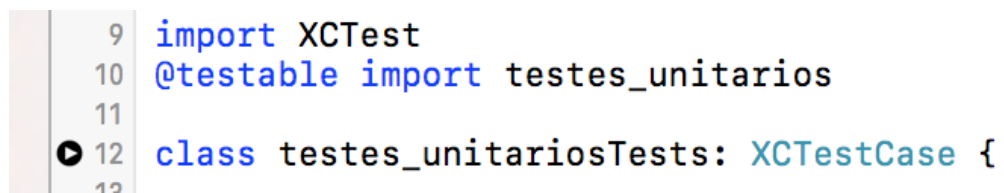
- Observe que a classe de testes precisa criar o ambiente necessário para ter acesso aos recursos que serão testados. Por isso foi criada uma instância da classe **ViewController**, para assim ter acesso à função a ser testada.
- Foram criados dois testes para validar ambas as condições *booleanas*, com o resultado esperado para cada uma das possibilidades.
- ATENÇÃO: Durante as atividades, você irá perceber que o Xcode leva um certo tempo para reconhecer as últimas alterações, indicando erros de compilação que já foram resolvidos. Basta aguardar ou executar a opção de menu **Product > Clean**.

Executando o Teste

Tudo muito bacana e bonito até agora, mas como rodar o testes e verificar se os resultados são os esperados?

Existem basicamente duas formas de se executar os testes unitários:

1. Executar todos os testes de uma classe de testes:
 - Essa opção é normalmente utilizada após uma alteração no código do projeto, que possa assim impactar no que já estava funcionando. É uma validação do que foi implementado.
 - Para executar todos os testes, você deve clicar sobre o botão que se encontra ao lado do nome da classe:



```
9 import XCTest
10 @testable import testes_unitarios
11
12 class testes_unitariosTests: XCTestCase {
13
```

2. Executar apenas o teste desejado.
 - Utilizado normalmente quando se está desenvolvendo os testes, e na realidade o objetivo é validar a própria função de teste.
 - Para executar apenas um teste, você deve clicar sobre o botão que se encontra ao lado da função de teste:

```

23
24 // Testes Unitários
25
26 func testIsNumberEven() {
27

```

Vamos lá então rodar o nosso teste:

- Execute o teste da função **testIsNumberEven()**
- Observe que o Xcode irá compilar o projeto e entrar em modo "Testing"
- Ao final ele apontará os resultados, que podem ser:
 - Test Succeeded - indicado pela cor verde
 - Test Failed - indicado pela cor vermelha
- Se a função testado está funcionando adequadamente e o teste foi construído de forma correta, tudo deve terminar na cor verde.

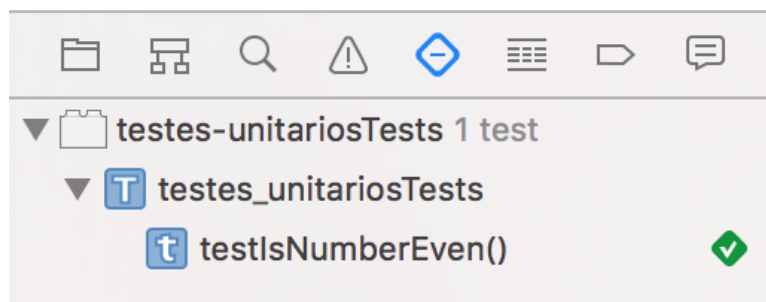
Um bom exercício aqui, a ser executado junto aos alunos, é "provocar" um erro, visando verificar se o teste identificaria uma situação de erro real. O erro pode ser provocado de duas formas:

1. Invertendo o retorno da função de origem.
2. Modificando o parâmetro da função de teste - por exemplo, colocando um número par ao invés de ímpar e vice-versa.

Verificando os Resultados

Além do feedback visual dos resultados dos testes na própria classe de teste, o Xcode proporciona outras formas de visualização, que são muito úteis, especialmente em projetos maiores, que irão conter centenas de testes, distribuídos em várias classes.

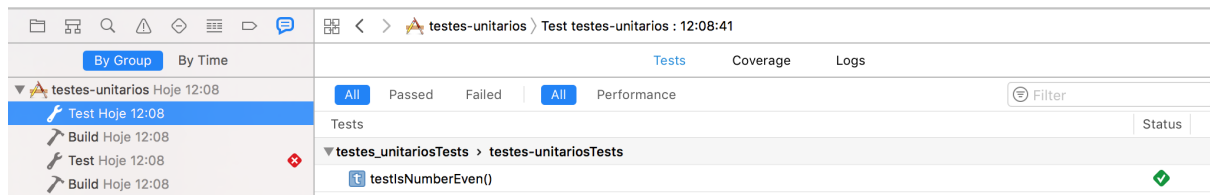
Inicialmente temos o painel **Test Navigator**, localizado na mesma barra de tarefas de navegação do projeto:



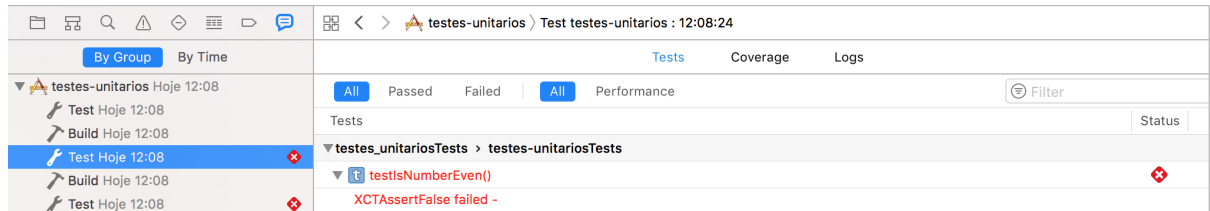
Neste painel teremos uma visão geral da quantidade de testes executados, os testes que foram bem sucedidos e os que falharam.

O Xcode proporciona ainda uma outra visão onde pode-se verificar com detalhes quais foram as falhas ocorridas. É o painel **Report Navigator**, onde pode-se consultar um log das últimas operações realizadas, incluindo os testes.

Ao selecionar um teste bem sucedido, o relatório irá trazer os detalhes na seção à direita do navegador:



Ao selecionar um teste com falha, o relatório irá trazer os detalhes do erro:



Atalhos para execução dos Testes

Assim como o atalho *command+B* compila o projeto e *command+R* executa, temos um atalho importante para execução dos testes unitários, que é o *command+U*.

Criando um teste um pouco mais complexo

Vamos criar agora uma função um pouco mais complexa, que trará a necessidade de um novo tipo de testes.

Retorne à **ViewController.swift**:

- Digite a seguinte função, para cálculo da sequência de Fibonacci:

```
// 2 - Função mais complexa - Sequência de Fibonacci
func sumEvenValueNumberFibonacciLimitSequence(limit: Int) -> Int {

    var sum = 0
    var a = 1
    var b = 1
    while b < limit {
        if b % 2 == 0 {
            sum += b
        }
        let h = a + b
        a = b
        b = h
    }
    return sum
}
```


Planejando e construindo o teste

Como testar uma função deste tipo, dado que não sabemos de antemão o limite de cálculo que pode ser passado a esta função?

Uma boa forma é definir um limite e testar se o valor de retorno é o esperado. Sendo, assim, iremos testar igualdade, não é mesmo. Para testar igualdade, utilizaremos o método **XCTAssertEqual**.

Um tipo de erro bem típico que o teste poderia identificar seria se o desenvolvedor, por engano `sum -= b` ao invés de `sum += b`, não é mesmo?

Acesse novamente a classe **testes_unitariosTests.swift**, e digite a seguinte função de teste:

```
func testSumEvenValueNumbersFiconacciSequence() {
    let limit = 4_000_000
    let answer = 4613732 // Para este limite, a resposta correta é: 4613732

    // Para forçar um erro, coloque um valor errado como resposta

    // 1 - Com a resposta padrão
    XCTAssertEqual(viewController.sumEvenValueNumberFibonacciLimitSequence(limit:
limit), answer)

    // 2 - Configurando a mensagem de erro
    XCTAssertEqual(viewController.sumEvenValueNumberFibonacciLimitSequence(limit:
limit), answer, "A resposta esperada é 4613732")
}
```

Temos uma novidade aqui. Além da linha de teste padrão (opção 1), é possível configurar a mensagem, em caso de erro (opção 2).

Temos um problema também. Seu código deve estar apontando um erro na referência à variável **viewController**, que foi definida dentro do escopo do teste anterior. Para que a variável fique disponível para todos os testes, vamos deslocá-la para o escopo da classe:

```
class testes_unitariosTests: XCTestCase {

    // Para ter acesso às funções da View Controller
    let viewController = ViewController()
```

Testando...

Para testar, comece pela situação de sucesso, em seguida provoque um erro. Consulte os painéis e veja todas as possibilidades.

Já temos dois testes agora. Rode tudo de uma só vez com o atalho *command+U*.

Teste salvando vidas (ou dinheiro)

Vamos analisar um pequeno estudo de caso, que mostra como um teste pode evitar perdas financeiras.

Considere que sua empresa lançou um aplicativo de treinamento online, que premia o cliente com moedas toda vez que um aluno finaliza um curso. Assim o aluno pode ir juntando moedas para, em algum momento, conseguir cursar um treinamento de graça. A regra de negócio estipulada pelo marketing, no entanto, prevê que o aluno deve receber em moedas no máximo 5% do valor do próximo curso.

Mas, por um engano do desenvolvedor, foi digitado na função de cálculo o valor 25%, e não 5%, como deveria ser.

Como os prazos estavam apertados, o time de desenvolvimento priorizou colocar o app na loja e deixou os testes unitários para um segundo momento. Infelizmente, para todos, a função que calcula o prêmio do cliente subiu sem ser devidamente testada, ou seja, subiu com erro. Após algumas semanas no ar, o time de marketing percebeu que o número de cursos cursados gratuitamente estava muito acima do previsto.

O pessoal de marketing chegou para o time de desenvolvimento e fez aquela pergunta clássica: "Vocês testaram a funcionalidade de premiação do cliente?"

Já ouviram falar de um caso assim? Pode acontecer na vida real, não é mesmo?

Construindo a função de cálculo

Novamente na classe `ViewController.swift`, vamos construir a seguinte função. Observe que o erro bomba relógio está inserido no código (valor 25, ao invés de 5):

```
// 3 - Um caso que pode gerar perda financeira
// Um app de cursos online premia o cliente com moedas sempre que uma lição é
completada
// O cliente deve ser premiado com moedas referentes a 5% do valor do próximo curso
func getLessonCoinAmount(coursePrice: Int) -> Int? {
    let lessonRewardCoinPercentage = 25
    let decimal = Double(lessonRewardCoinPercentage) / 100.0
    return Int(Double(coursePrice) * decimal)
}
```

Antes tarde do que nunca

Assim que foi informado do erro, o time de desenvolvimento criou, em minutos, a seguinte função de teste, na classe `testes_unitariosTests.swift`. Observe que, no teste, a regra de negócio está sendo seguida corretamente (valor 5):

```
func testCoinRewardAmount() {
    let coursePrice = 100
    let coinsRewarded = 5
```

```
XCTAssertEqual(viewController.getLessonCoinAmount(coursePrice: coursePrice),
coinsRewarded, "Existe um erro no cálculo da premiação do cliente")
}
```

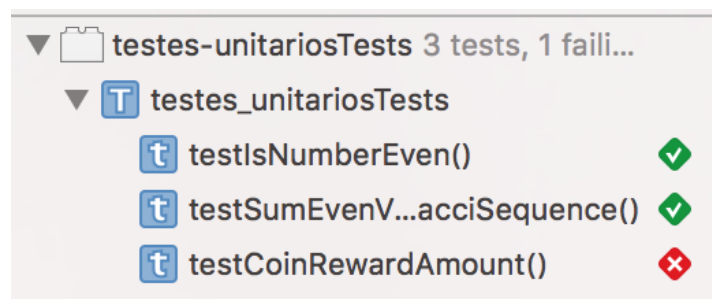
Ao rodar o teste, para a surpresa de todos, o erro foi apontado, em todas as visualizações:

Na própria função:

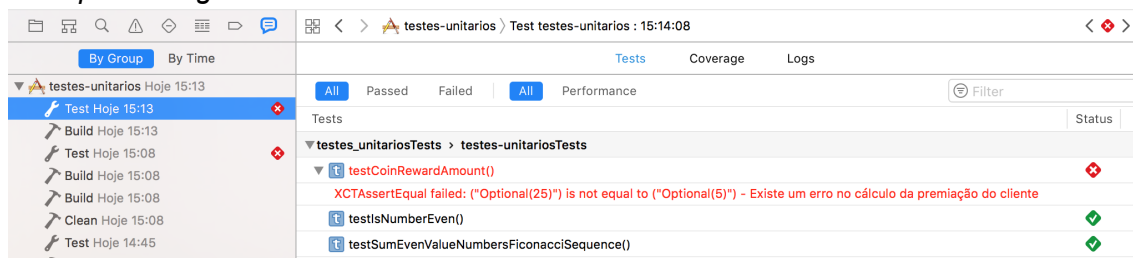
```
54 func testCoinRewardAmount() {
55     let coursePrice = 100
56     let coinsRewarded = 5
57
58     XCTAssertEqual(viewController.getLessonCoinAmount(coursePrice: coursePrice), coinsRewarded,
59                     "Existe um erro no cálculo da premiação do cliente")
60 }
```

XCTAssertEqual failed: ("Optional(25)") is not equal to ("Optional(5)") - Existe um erro no cálculo da premiação do cliente

No Test Navigator:



E no Report Navigator:



O benefício do teste unitário

Um teste com 3 linhas de código poderia ter evitado uma perda financeira para a empresa, que às vezes é incalculável.

Estão convencidos agora sobre a importância dos testes unitários? É por isso que eles são obrigatórios em ambientes de desenvolvimento corporativos, assim como em provas de seleção de novos colaboradores.

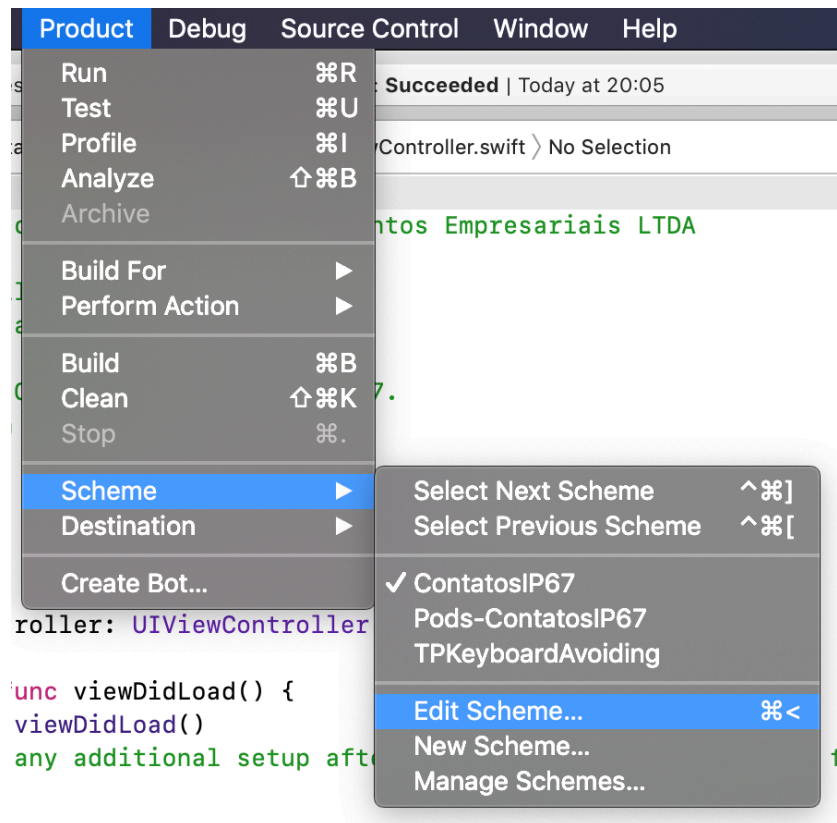
Cobertura de Código

Cobertura de código é uma ferramenta no Xcode que é usada para obter informações sobre quanto do seu código você está testando com o seu conjunto de testes. Ele informa exatamente quais partes do seu código foram executadas durante um teste e quais partes do seu código não foram. Isso é extremamente útil porque você pode tomar uma ação focada com base nas informações fornecidas pela Cobertura de código.

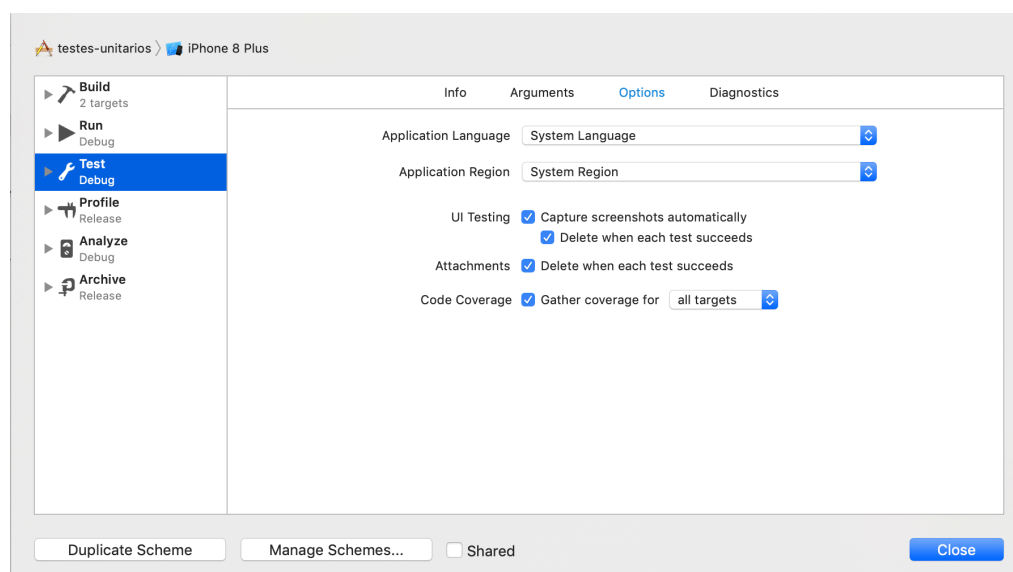
Algumas empresas utilizam a cobertura de código como parâmetro para aceitar uma nova implantação, somente se a cobertura de código atingir um percentual mínimo.

Para habilitar o recurso de cobertura de código, ou *Code Coverage*, é necessário editar o esquema de compilação através da opção de menu (**Product** | **Scheme** | **Edit Scheme**).

Veja:

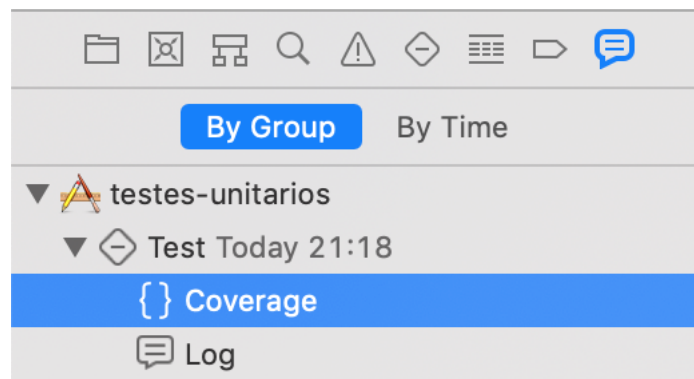


Selecione a ação *Test* e certifique-se que o campo *Gather Coverage* esteja habilitado, como na tela a seguir:



Depois desta configuração, feche o editor de esquema e execute seus testes. Desta vez, o Xcode irá monitorar quais partes do seu código foram executadas durante este teste, e quais partes não foram. Essas informações vão indicar quais partes do seu código poderiam usar mais alguns testes.

Para ver os dados de cobertura, abra o navegador Relatório (*Report*) na barra lateral esquerda no Xcode. O ícone mais à direita nesta barra lateral representa o navegador do Relatório:



Esta tela mostra alguns relatórios resultantes dos testes. O relatório que nos interessa é de cobertura de testes - *Coverage*. Ao selecionar essa opção, o relatório será exibido na área de edição do Xcode. No relatório você pode visualizar todos os arquivos do app, assim como o percentual de cobertura de testes em cada arquivo.

Veja o relatório após a construção de todos os testes:

testes-unitarios > Test > {} Coverage	
<input type="checkbox"/> Show Test Bundles <input type="text" value="Filter"/>	
Name	Coverage
▼ testes-unitarios.app	<div><div></div></div> 68,85%
▼ AppDelegate.swift	<div><div></div></div> 33,33%
testes_unitarios.AppDelegate.application(_:UIApplication, didFinishLaunchingWithOptions: [UIApplication.LaunchOptionsKey]?) -> ()	<div><div></div></div> 100%
testes_unitarios.AppDelegate.applicationWillResignActive(_:UIApplication) -> ()	<div><div></div></div> 0%
testes_unitarios.AppDelegate.applicationDidEnterBackground(_:UIApplication) -> ()	<div><div></div></div> 0%
testes_unitarios.AppDelegate.applicationWillEnterForeground(_:UIApplication) -> ()	<div><div></div></div> 0%
testes_unitarios.AppDelegate.applicationDidBecomeActive(_:UIApplication) -> ()	<div><div></div></div> 100%
testes_unitarios.AppDelegate.applicationWillTerminate(_:UIApplication) -> ()	<div><div></div></div> 0%
▼ ViewController.swift	<div><div></div></div> 87,5%
testes_unitarios.ViewController.viewDidLoad() -> ()	<div><div></div></div> 100%
testes_unitarios.ViewController.didReceiveMemoryWarning() -> ()	<div><div></div></div> 0%
testes_unitarios.ViewController.isNumberEven(num: Swift.Int) -> Swift.Bool	<div><div></div></div> 85,71%
testes_unitarios.ViewController.sumEvenValueNumberFibonacciLimitSequence(limit: Swift.Int) -> Swift.Int	<div><div></div></div> 100%
testes_unitarios.ViewController.getLessonCoinAmount(coursePrice: Swift.Int) -> Swift.Option	<div><div></div></div> 100%
testes_unitarios.ViewController.discountCalculator(price: Swift.Double, percent: Swift.Int) -> Swift.Double	<div><div></div></div> 100%

No relatório podemos observar o percentual de cobertura de testes em cada método. O tamanho da barra indica quantas linhas de código foram executadas durante os testes.

O relatório de cobertura é um ótimo indicador se você ou sua equipe estão minimamente escrevendo testes para garantir a qualidade de seu código. Porém não tente perseguir os 100% de cobertura. A maioria das empresas trabalha por volta de 70% de cobertura como um índice aceitável de qualidade, até porque não é 100% do código que precisa ser testado.

Exercício

Dada a função a seguir:

```
// 4 - Exercício
// Discount Calculator
// Recebe preço original e percentual de desconto
// Retorna preço com desconto
func discountCalculator(price: Double, percent: Int) -> Double {
    return price - (price * (Double(percent) / 100))
}
```

Construir uma função de teste para validar se o cálculo de desconto está sendo efetuado da forma correta.

Referências

Os exemplos desta apostila foram inspirados no blog de desenvolvimento em iOS *DevsLopes*. O link com o tutorial está no endereço https://www.youtube.com/watch?v=-VdQfX6q_a8